Service Oriented Architectures

Phillip J. Windley
phil@windley.com

Service-oriented architectures (SOAs) are a particular kind of software architecture that is designed to create a dynamically organized environment of networked services that are composable and interoperable. The fundamental building block of an SOA is a service and services are composed in specific ways to create applications. SOAs separate the services from their implementation using the notion of an interface. This interface creates a contract about how the interaction between the parties will proceed. SOAs provide a number of benefits for creating federations of services among disparate and loosely connected organizations while allowing each organization to maintain its autonomy in terms of how it builds and designs services as well as their ownership.

Traditional Application Architectures

To understand service-oriented architectures, let's look first at a traditional application architecture. To make this more concrete, consider the example of an agency that is responsible for licensing cars and trucks. As part of its responsibility, the agency desires to offer a Web application to vehicle dealers that allows them to license a vehicle at the time it is purchased on behalf of their

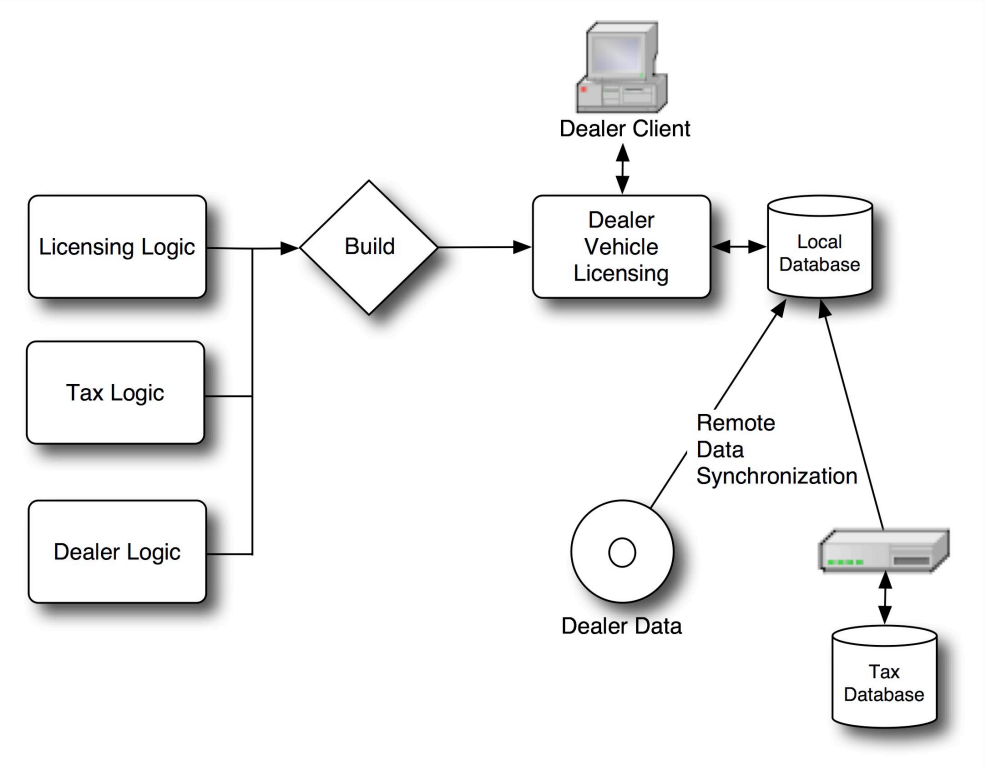

Figure 1: Traditional Application Architecture

customer. The application will also allow the dealer to calculate and pay taxes due on the vehicle.

In a traditional application architecture, the application builders would be responsible for linking the business logic for managing dealers, performing the licensing, and calculating and collecting taxes into a single application. The algorithms would need access to data about tax codes and valid vehicle dealers, but since these are managed by other agencies, this data would have to be synchronized to the local data store on a schedule that ensures that its not out of date. Figure 1 shows schematically how this application might be built and deployed and how the data would be synchronized.

This model presents the licensing agency with several problems.

1. The agency building the application is in the business of licensing vehicles and understands the business processes associated with that task very well. They are not, however, in the business of regulating dealers or collecting taxes. Nevertheless, in this model, they will have to put business logic associated with both of those tasks into their application. In the most likely scenario they would write code that approximates the real processes close enough for their purposes since reusable code that performs these tasks is not available.
2. The data that the agency needs to run the application is owned and maintained by other agencies. This data may change daily or ever hourly. Operating the licensing application thus requires that the licensing agency synchronize the data from the dealer and tax agencies often. This synchronization may take place over a network on a batch basis or even happen via physical media such as CD-ROM or tape. Keeping the data synchronized and accurate presents a significant management challenge to the license application operators.

Applications as Integrations of Services

Now, consider a service oriented architecture. In an SOA-based application, the licensing agency can concentrate on what it does licensing. The application designers build business logic concerned with licensing and the local data store contains data about vehicle licensing. The data and business logic associated with the tax and dealer functions would be built and operated by the agencies responsible for those functions and presented to other agencies as a service that can be accessed as needed. Figure 2 shows schematically how such an application would be built and deployed.

This model has several advantages:

1. Each agency is responsible for building and operating applications associated with the business processes that they own.
2. Each agency maintains its own data. Since data is never synchronized, it is not out of date or inconsistent.

3. Multiple agencies may make use of the tax or dealer services. This keeps each from building the business logic on their own and ensure that each is using the same business logic.

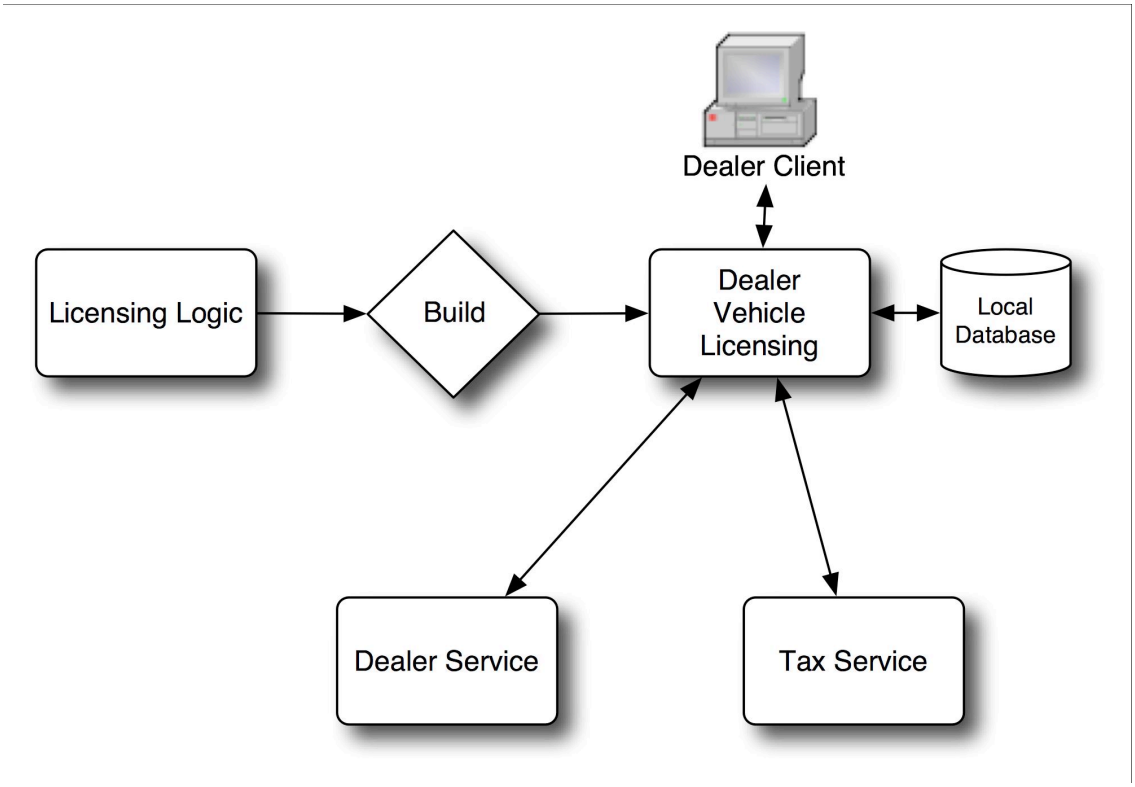

Figure 2: SOA Application Architecture

This model also presents some additional responsibilities for each agency. Let's take the tax agency as an example. Previously the tax agency was only responsible for shipping off the relevant data on some scheduled basis. Now, they are responsible for creating and operating a service that meets the needs of the licensing agency along with any other agency using the service. The service must be accurate, reliable, secure, and scalable. Of course, what they get in return is accurate tax collection and a knowledge that their data is up to date and canonical.

Document Metaphor

Service oriented architectures differ from distributed architectures in the following important way:

Distributed architectures are based on a notion of complex endpoints and transport carrying simple data. Early distributed system designers were creating a programming style that used hardware as a model for software development. Hardware is characterized by complex components, chips that hook together in simple ways. The data that is transported between chips is mere bits and even when aggregated into busses rarely looks anymore complicated than an array of bits. This model influenced the designers of early distributed systems to create complex, rigid interfaces that carried simple, serializable data.

Service oriented architectures are characterized by simple endpoints and transport carrying rich data. SOAs, in contrast, are focused on simple interface models that exchange rich, structured data. Current implementations in XML can carry complex data. Machine readable documents are used to publish interfaces to services and these documents are carried between endpoints to affect overall system behavior.

This concept is called the *document metaphor*.

SOA Model

Properties

Service oriented architectures are characterized by the following interrelated properties.

Discoverable and Dynamic – Services are discovered using a directory. Services are bound at runtime, rather than compile time.

Loosely Coupled – SOAs are composed of multiple services connected in such a way as to be resilient in the face of network failures and latency. This *loose coupling* gives SOAs a distinctly different architecture than programs that are distributed, but still connected synchronously and in ways that make the overall system brittle.

Locationally Transparent – SOAs are constructed in such a way that the overall system is unaware of, or at least ambivalent to the location of various services.

Diversely Owned - SOAs may be composed of services which are owned and operated by outside organizations. *Diverse ownership* implies that the published service interface will be treated as a blackbox from the standpoint of the programmers since they cannot penetrate the interface and modify code and behavior behind it.

Interoperable – Standards ensure that services from differing organizations can use each other's services.

Composable – Applications in SOAs are created by composing pre-existing, well-tested services from multiple providers.

Network-addressable – Networks are central to the idea of services that are discoverable and interoperable. This allows applications to be composed that run on different machines.

Self-healing – When applications are created by composing dynamically discovered components that are owned by multiple organizations, the ability of the system to rediscover and bind to working services when services fail is critical.

The properties that define SOAs have some significant consequences:

Latency is a fact of life in SOA-based implementations. This follows from locational transparency and diverse ownership and drives loose coupling.

Loose coupling is accomplished through asynchronous communication between components. Asynchronous design is different from traditional systems design and can be more challenging and requires different debugging skills.

Loose coupling and diverse ownership mean that interoperability and strong standards are crucial to successful SOA deployment.

The Internet serves as a universal information bus between services. Specialized protocols like DCOM, CORBA, and RMI have been used in many distributed systems, but diverse ownership and locational transparency require an information transport bus that is more universal. HTTP, SMTP, and other Internet protocols reach everywhere and are universally understood.

The Players and Actions in a Service Oriented Architecture

A service oriented architecture has three component roles: client, service provider, and service broker. This section describes those roles and the actions they take in relationship to each other.

Clients

Clients are the requestors of services. A client is typically a software application or service that requires a service. Clients make requests of service providers.

Service Providers

Service providers publish services and make them available to clients. Service providers accepts client requests and exectute them. The service provider is some software application or service running on a network addressable computer.

Service Brokers

Brokers bring clients and providers together by providing a registry of known services, along with the service contract (interface) so that clients can pick the correct service. Brokers take advantage of the locational transparency of SOA-based implementations and connect clients to any one of a number of service providers who offer a given service contract.

Actions: Publish, Discovery, Bind and Execute

As shown in Figure 3, clients initiate a service request by contacting the service broker and using the broker's directory service to discover a service that has a specific interface contract. The broker knows which service providers support which interfaces because the service providers register with the broker by publishing their supported interfaces. Once the broker returns the information about the service provider, the client uses that information to bind to a particular service at a service provider and sending it a request to execute that service.

Publish and discovery are two important ways that SOAs differ from more tightly coupled architectures like those created with Enterprise Java beans. To use EJBs you need to know where the service is and you have to conform, in a

manual way to the interface rather than automatically adapting as you can from a self-describing, published interface like those found in SOAs.

Published interfaces differ from public interfaces. Public interfaces have a public design, but are only known by known clients and thus can be changed because all of the clients can be found and changed. Published interfaces might be used by clients without the service operators knowledge or express permission. This means that the contract represented by the interface cannot be easily changed. One way to envision this is to think of the services in an SOA-based system in the same way you think of documents on a Web server. Because each document on the Web server has a URL, a distinguishing name, other documents can link to it without the express permission or knowledge of the document's owner. This creates a situation where URLs cannot be changed easily because they represent a *published* reference to a document.

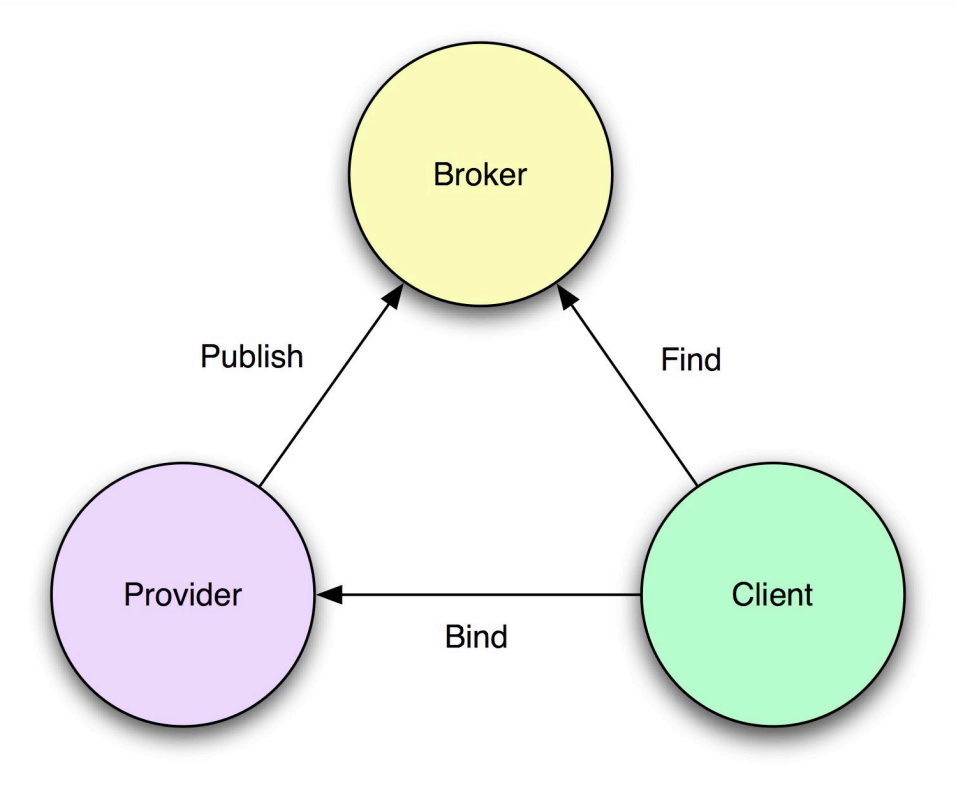

Figure 3: SOA Service Triangle

Messaging and Message Exchange Patterns

Service oriented architectures are based on the idea that clients make requests of services and receive responses from services via messages. Designing an SOA-based application requires making choices about the messaging style, applicable messaging patterns and message transport.

Messaging Style:

Synchronous

Figure 4 shows a schematic representation of a synchronous message exchange. The client sends a request to the service and then blocks. Meanwhile, the service completes its processing and sends back a response, allowing the client to unblock and continue.

Synchronous messaging has a number of advantages. Synchronous message passing is relatively easier to develop and debug than asynchronous messaging. Programmers are used to thinking of processes synchronously and often miss subtle bugs associated with asynchronous processing. Transaction processing is also easier to do synchronously. However, synchronous messaging leads to tight coupling and, thus should be avoided when latency is an issue. This makes it more appropriate for accessing services on the local network, or in other tightly controlled circumstances.

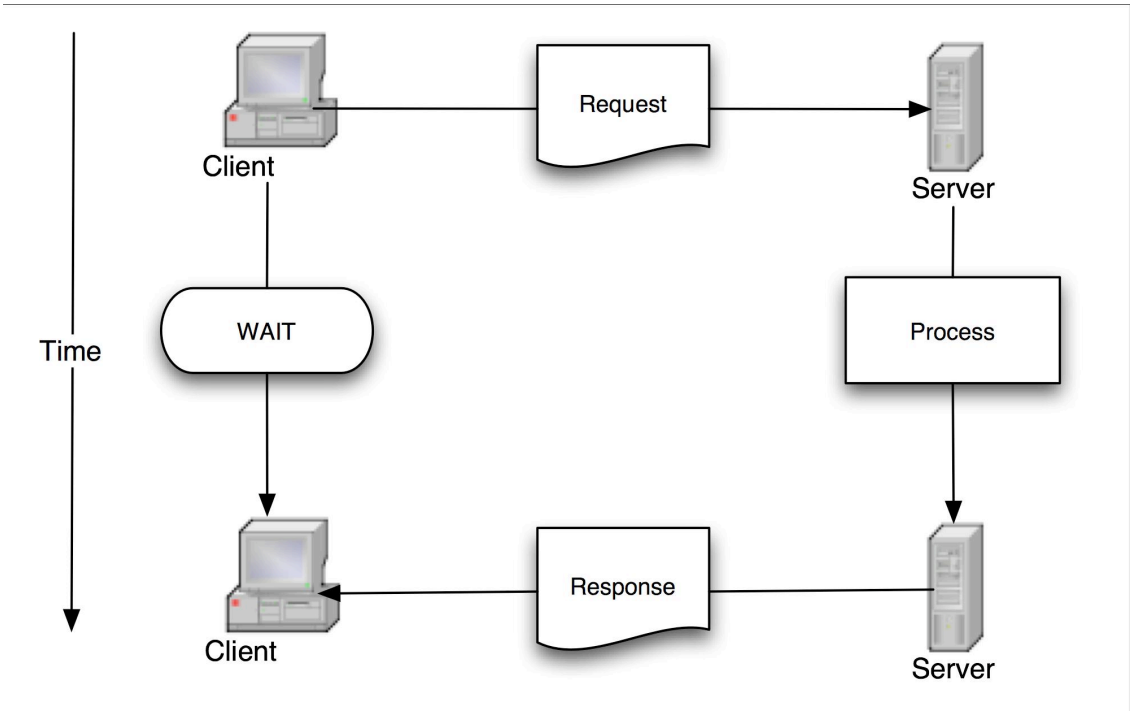

Figure 4: Synchronous Messaging

Asynchronous

Figure 5 shows a schematic representation of an asynchronous message exchange. The client sends a request to the service and receives an immediate acknowledgement that the message has been received, but nothing else. The client can continue on with other work while its waiting for the service to complete its processing. When the service finishes, it sends the response to a message queue where it waits to be picked up by the client.

The advantages of asynchronous message passing include

- Loose coupling between sender and receiver—if latency increases due to network or processing congestion, for example, the client is designed for an indeterminent response time.
- Does not block sender—the client is free to do other processing while the request is being processed.
- Network does not need to be available since messages can be queued.

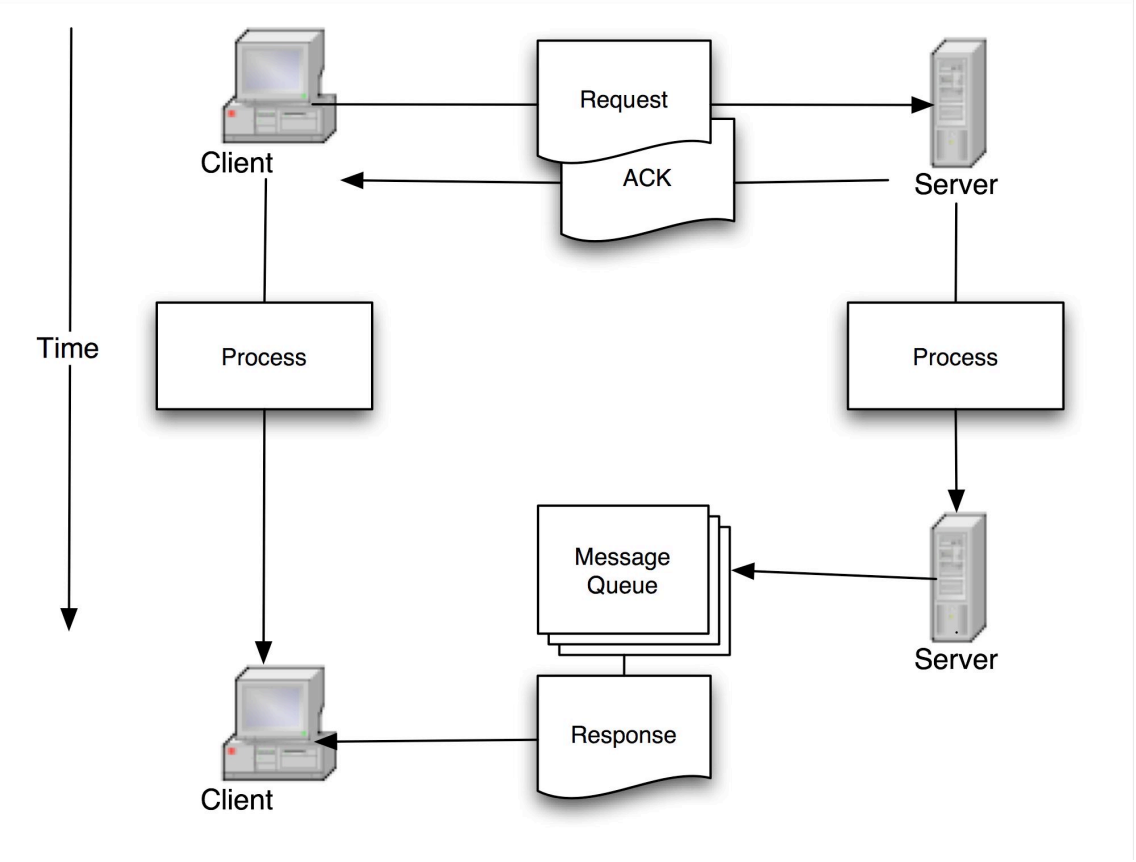

Figure 5: Asynchronous Messaging

Message Correlation

In a synchronous system, it is easy to tell which response goes with which request because they are interleaved with a response always following immediately after the corresponding request. In an asynchronous exchange, there may be multiple requests made and multiple responses and they might be interleaved in any number of ways. Correlation is the process by which a messaging system determines which responses go with which requests. This can be done using tokens attached to requests that the service guarantees it will return unchanged so that the client can correlate responses to requests.

Message Delivery

A related concept to message correlation is message delivery. An asynchronous messaging system can be set up to provide

- Guaranteed delivery
- Only-once delivery
- In-order deliver

To see why these are important, consider a bank account service taking requests from clients for deposits, withdrawals, and transfers. In this example, guaranteed delivery is important since clients will expect that once they've told "the system" to transfer money, for example, that its been done.

Sometimes network or system errors can result in duplicate messages (or messages that are sent twice). Only-once delivery ensures that a request won't be delivered multiple times. Thus, a message to make a withdrawal will only be processed once.

In-order delivery ensures that a series of requests are processed in the same order that they were sent. So, if your bank account has $1000 and you sent a message to deposit $1000 and then a message to transfer $1500, the deposit would be guaranteed to happen first, avoiding an overdraft.

Popular Patterns

Messages are exchanged in several recurring patterns. This section discusses three of the most common: request/response, publish and subscribe, and broadcast/multicast.

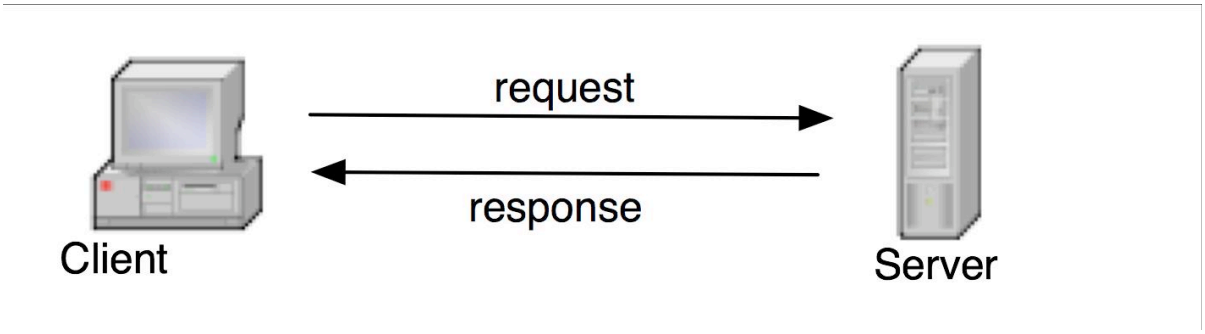

Figure 6: Request Response Pattern

Request/Reponse

Request/response is the simplest and most widely used messaging pattern. In a request/response pattern that client and the service talk to each other directly and as peers. Figure 6 shows a schematic representation of the request/response message pattern. Request/response can be set up to use either synchronous or asynchronous messaging, but synchronous is by far the more common approach.

Publish and Subscribe

In a publish and subscribe pattern, the service provides a set of topics that it creates messages about and an interface where clients can subscribe to a topic. When the service has a message for a particular topic, it sends the message to each subscriber for that topic. Figure 7 shows a schematic representation of a publish and subscribe message exchange pattern. Note that the client only notifies the service once that its interested in a particular topic, but may receive multiple messages on that topic from the service.

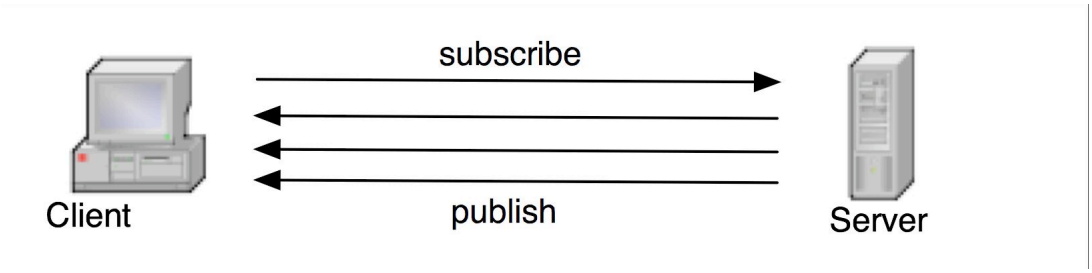

Figure 7: Publish and Subscribe Pattern

Broadcast and Multicast

Broadcast, as its name implies sends a message to every device or service in a particular domain. The message is usually identical for every recipient. Typically, no response is expected, although that is not always the case. Figure 8 shows a schematic representation of a broadcast message exchange. Multicast is like broadcast except that it sends the message to only a specified subset of the domain.

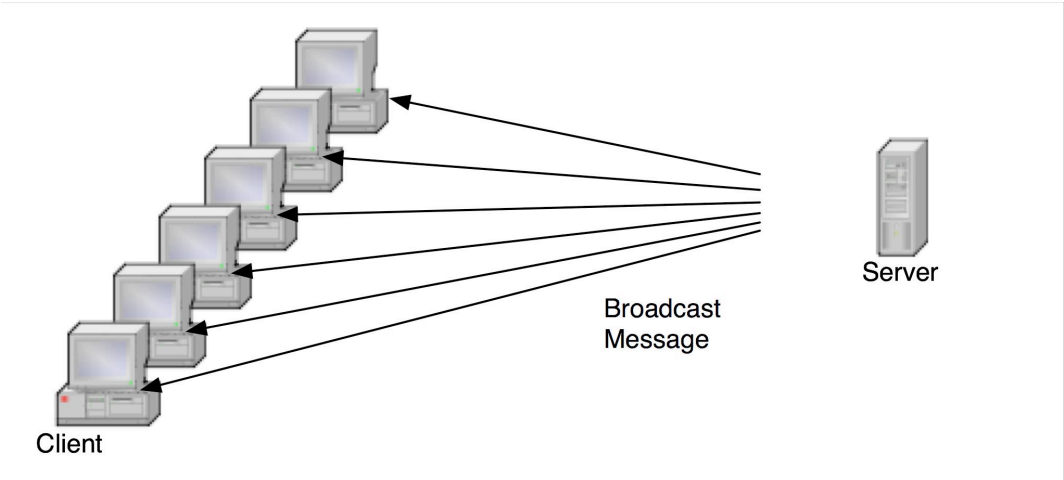

Figure 8: Broadcast and Multicast Pattern

Building Blocks

Intermediaries

Because they have standardized, interoperable interfaces, service oriented architectures naturally lend themselves to being augmented by services that proxy the original service and add some new service or feature on top of the original. These proxies are called active intermediaries.

Active intermediaries store and forward messages from one Web service to another in a reliable way. Along the way, they can filter, transform, log, and analyze the message flow in just about any way you can imagine. This is an advantage over a directly connected model because active intermediaries can add security, reliability, availability, scalability, and interoperability to a service without modifying the service itself.

To see how this is possible, take the simple example of authentication and authorization. Suppose you've built a program and made it available as a service. Later you decide to restrict access to your service to a small collection of trading partners. You could modify the service itself, but this makes it less general, and thus harder to reuse in some other capacity. An active intermediary can sit in front of your service and perform the authentication and authorization using LDAP, SAML, or some other system—all without modification to the original code.

The authentication and authorization proxy is an example of context-sensitive filtering. The active intermediary is filtering the messages that are allowed to pass to the Web service based on a particular context, in this case the authority

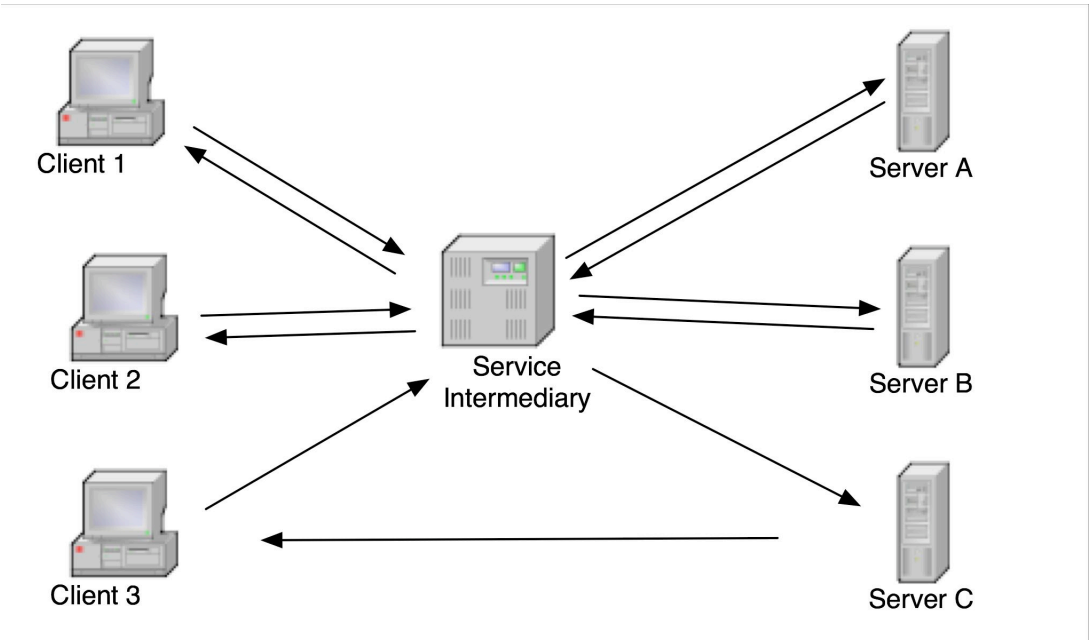

Figure 9: Service Intermediary

of the agent making the request.

Because they can look inside the messages and make decisions about what to do next, active intermediaries create networks of services. Messages are routed through this network. A message representing a purchase order with enclosures written in German may need to go through a language translation service, for example, before its routed to the Chinese supplier.

A number of vendors supply active intermediary products that work with Web services-based SOAs. Using these intermediaries in a design mitigates some of the concerns that arise surrounding changing standards, security, and complex deployments.

Aggregators and Orchestration

Aggregators provide a composite service based on a collection of other services. Services oriented architectures are defined in terms of this ability to aggregate services, but it is possible to build aggregators, special applications that are used specifically for orchestrating the composition and workflow of the aggregated services.

Figure 10 shows a schematic representation of how an aggregator works. The client makes a request of the aggregator who fulfills the request by making requests, in turn, of services A, B, and C.

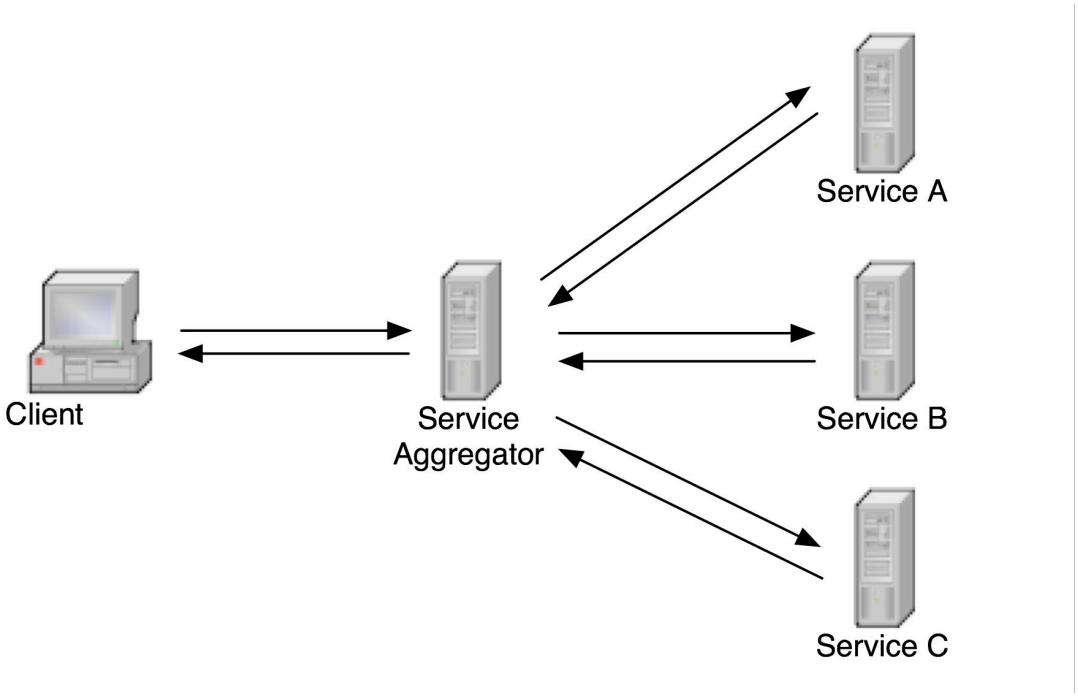

Figure 10: Service Aggregator

Aggregators and orchestration is important in a complex SOA-based application because building the required workflow logic into each service, an external agent

executes the workflow. Using an aggregator helps to preserve the reusability of the individual services because they can remain more general.

Identity and Security

Service oriented architectures create unique requirements for identifying who is requesting services and securing them from unauthorized use. Traditionally organizations have only had to manage the identities of the their employees and a few outside partners. What's more, perimeter defense in the form of firewalls and hardened servers has been the primary means of protecting the digital assets of the enterprise. An SOA-based application might access services in many different organizations. The application is not just requesting data through a query, but actually requesting that some function be executed on its behalf. This presents a whole new set of challenges to enterprises deploying SOA-based applications.

Rather than managing resources at the user, server, or network level, SOA-based applications require that resources be managed as services, records, documents, or even fields. Managing identity at this level implies that the organization has a business architecture that can define the access control policies for each resource and an efficient means of putting the policy into practice once its created. Of course, since resources are in a constant state of flux, the policy needs to be resilient and maintained.

Data in the Service Oriented Architecture

Databases are often treated as a place to persistently store variables for a particular application rather than a collection of resources that can be used by multiple applications. Designing data sources so that they can be used by multiple applications is consistent with the intent of service oriented architectures.

Even though we may not know *a priori* exactly how that data will be used, there are principals we can follow that will help in the design of generally useful data stores. These principals are designed to ensure that data sources will be useful in a wide variety of circumstances, not just those for which they were initially designed. For the most part, adhering to these principles should not appreciably increase the cost of creating the data source.

1. Every data record and collection is a resource. This principal reiterates the idea that the data has an importance independent from any specific application. Collections of resources are the results of queries and represent a resource as well.
2. Every resource should be universally uniquely addressable on the network. The standard way of creating universally unique addresses is to use a URI. Giving each resource and collection of resources a unique address ensures that each application can find and use the resource over time and refer to it in a way that lets other applications find the same data.
3. The address for a given resource should remain the same as long as the resource exists. For the address to be useful, it has to be consistent over

time.

4. Data queries should be done in a way that is locationally transparent and supportive of caching where appropriate. When this principal is followed the data resources can be replicated and results can be cached for reliability and performance.
5. The format and style of a data query should be documented in a structured way so that other applications can determine how to use it. Just as services in an SOA should be self-describing, so should the format of the data source be self-describing to aid in its use by other applications.
6. Data returned from a query should preserve the structure of the original data as much as possible. The structure of the data is important to further use. A classic example of breaking this principal is to format the results of a query as HTML rather than preserving the structure so that another program can easily use it.
7. Meta information about the structure should be available online to describe the structure. The structure of the data should be well documented and the documentation, in machine readable format, should be available online.
8. Provision should be made to translate data from one structure to other useful structures and presentation styles. Because the structure of the data is maintained as long as possible, it can be translated automatically into other useful structures as required. For example, it could be reformatted into HTML for presentation to a browser as a composable step to the data query.
9. Metadata describing the data (over and above its structure) should be widely available online. Any useful metadata about the data should be available either with the data or separately as is appropriate. This metadata should be available online and in a machine readable format for use by applications using the data.
10. Data should be returned in standard structures, where such exist. There are a number of industry standards for structured data of one kind or another. Using standard forms for the structure of the data increases the interoperability of the data with other applications.

Principals of Loose Coupling

As we have mentioned, loose coupling is one of the most important features in a service oriented architecture. Loose coupling is a necessary condition in SOA because they are designed to function in the face of high latency and even outright network failure. Furthermore, since services making up an application can be owned and operated by many different organizations, loose coupling prevents unnecessary dependencies which would make such an application unmanageable.

Table 1 compares tight coupling and loose coupling for a number of different properties. The comparison gives a good overview of how some architectural decisions lead to tight coupling.

	Tight Coupling	Loose Coupling
Interface	Class and Methods	Fixed verbs (i.e. RESTian)
Messaging	Procedure Call	Document Passing
Typing	Static	Dynamic
Synchronization	Synchronous	Asynchronous
References	Named	Queried
Ontology (Interpretation)	By Prior Agreement	Self Describing (On The Fly)
Schema	First-order	Higher-order
Communication	Point to Point	Pub & Sub /Multicast
Interaction	Direct	Brokered
Evaluation (Sequencing)	Eager	Lazy
Motivation	Correctness, Efficiency	Adaptability, Interoperability
Behavior	Planned	Adaptive
Coordination	Centrally Managed	Distributed
Contracts	By Prior Agreements, Implicit	Self Describing, Explicit

Table 1: Tight vs. Loose Coupling

I view this table as a continuum rather than a black or white distinction. Few implementations will be in the “loosely coupled” column on every issue. These architectural choices will be determined by many things. A particular service implementation might pick a feature from one column or the other, mix and match as it were, to get a desired result and the completed system will be more or less loosely coupled depending on those choices.

In addition to the architecture choices listed above, there are a number of implementation choices that affect the loose coupling of systems. The following implementation principals will help create a loosely coupled implementation:

Avoid changing or extending the interface methods. Course-grained systems have a small number of verbs and they should remain unchanged as much as possible. HTTP is a perfect example of an interface with a small set of fixed verbs (i.e. GET). The HTTP interface has had only two version changes in over a decade of use.

Control change by using a dictionary interface. When a fixed set of verbs will not work, a dictionary interface provides a set of verbs for finding and executing right method. This is akin to data-driven programming in the Lisp world.

Calls should return documents not objects. The issue here is largely one of granularity. In a loosely coupled system, where latency is a real issue, getting back a pointer to an object isn't very useful.

Avoid binary compatibility. Systems that require binary compatibility aren't loosely coupled since an upgrade on one end requires an upgrade on the other.

Don't confuse an API with a contract. This emphasizes the difference between a protocol and an API. A protocol is a sequence of document exchanges that is required for compatibility. An API is more of a one-way declaration of how the methods work. Protocols are a more useful metaphor in a document-based SOA.

Version the contract. Versioning can be difficult to do. The task is made easier using intermediaries . If you don't version, you're back to the tightly coupled upgrade issue again.

Don't build an API for data transfer. The point here is pretty simple: we already have an API (protocol) for data transfer. Its called HTTP. Don't invent another one.

Benefits of SOAs

Service Oriented Architectures offer several significant benefits:

Code re-use - Creating a service layer on top of the business logic requires the design and documentation of a complete interface to the service. Encapsulating the code in an interface enables its use beyond the first application for which it is developed. This has several important consequences:

Return on Investment - Code re-use increases the return on investment in the original code because it can be used in multiple places.

Correctness - Code re-use increases functional correctness because the code is subject to more users and diverse applications. Because it is a service, there's no need for patches or code releases. Once its fixed on the server, the corrected service is available to all. Service versioning is necessary to ensure that downstream services that may be relying on incorrect functionality can be updated and tested with the new version of the service outside of the production environment.

Maintainability - The design and documentation of a service interface creates a barrier that has several consequences to the maintainability of the code in the system.

Productization - The service itself can be treated as a *product* with attendant version control, bug tracking, etc. The architecture is likely to be well thought out in connection with the interface. The rest of the system is using a well defined interface to the service.

Security - The interfaces that define the service layer can be proxied by other applications that provide authentication and authorization services in a more general manner. These proxies provide identity services for the service layer in a consistent way. This means that security code is kept out of the business logic, increasing its capability for re-use and increasing the likelihood that security is done correctly and in compliance with enterprise policy.

Focused Developer Roles - The interfaces focused on business operations gives developers a chance to focus on specific areas of develop in two dimensions. First, because service interfaces map to specific business domains, developers have the opportunity to specialize in particular domain areas and understand them well. Second, a layered architecture means that developers can specialize in the technologies and programming practices that support each layer such as database technologies, application and business logic, service layer development, application assembly, and presentation layer, to name a few.

Better Alignment with Business Goals - Service interfaces are focused on business operations. An SOA is created in terms of logical business operations and the services needed to support it

Configuration-based Application Development - In an SOA, applications are assembled from services and other code that is written to augment the services and provide specific functionality. This assembly can increasingly be done using workflow engines and other tools that allow the assembly to take place through configuration rather than traditional programming.

Feature Augmentation - SOAs lend themselves to having proxies placed in front of the service interfaces so that the features of a service can be augmented. For example, an authentication and authorization service may be placed in front of a service to add authentication and authorization functionality to the base server. Another example is the augmentation of a service with a logging proxy that provides strong logging mechanisms for service level auditing.

Better Scalability - the property of locational transparency means that service clients can't tell which machine is servicing a request. This means that many service requests can be processed in parallel.

Technology Interoperability - strong standards for interoperability between services mean that the particular technology used to create a service, which programming language it was written in, and what operating system it is on have less importance than in more tightly coupled architectures.

Potential SOA Missteps

Service-oriented architectures have significant advantages, but there are some areas that require special care and attention in a SOA:

High Availability. For an SOA to provide high availability services, the service broker (including the service directory), and the message delivery system (transport) must be reliable and trustworthy.

Sessions / Transactions. Database sessions and transactions can introduce tight coupling into the system. To avoid this, maintain an abstraction level for services above the level of database sessions and transactions. Multiple services should not share a database session or transaction.

Versioning. Services should be versioned to support change management activities and allow users to reduce the need for lock-step upgrades among service providers and clients.

Performance Instrumentation and Documentation. Services should be instrumented so that clients can determine where a request is waiting and performance statistics surrounding requests of a particular type. This can aid service assemblers in knowing what kinds of service levels to expect.

Remain Course Grained. Be careful to maintain a reasonable amount of work in a service. Services that are too fine-grained create performance bottlenecks and do not handle high latency situation well.

Environments. Determine essential environment settings for the service like character-set (Unicode) and time parameters including time zone and daylight savings if appropriate.

Logging / audit. High latency environments based on asynchronous messaging and using services outside the firewall can be difficult to debug. Services should be built so that they log their state and session information in such a way that error detection becomes manageable.

Implementing Service Oriented Architectures with Web Services

So far, in this document we have avoided the mention of specific technologies, focusing instead on the properties and uses of service oriented architectures. One can develop a system that has an SOA without any special help, but that requires developing a significant amount of overhead to support he publish, find, and bind functionality we've discussed. Alternatively, there are several technologies that can be used to create SOAs.

One such technology is called Jini and provides a complete set of libraries in Java that support building SOA-based applications. The technology that has gained traction, however, and is garnering most of the attention goes by the name "Web services." This is an unfortunate name that is constantly confused with the more general use of the term to refer to any application delivered over

Web. One way to define Web services is as self-contained pieces of code that have three distinguishing properties:

1. They communicate in an interoperable XML protocol, such as SOAP.
2. They describe themselves in an interoperable XML meta-format, such as WSDL.
3. They are able to federate globally through XML based registry services, such as UDDI.

These three properties and the protocols that implement them achieve the bind, publish, and find functionality of an SOA.

SOAP

SOAP, or the Simple Object Access Protocol, allows a client and server to exchange requests and responses, performing the bind function of the SOA model. SOAP is an XML-based protocol that describes how structured and typed (usually XML) messages can be exchanged between multiple network nodes. SOAP defines an <Envelope/> element that contains an optional <Header/> element and a <Body/> element. The <Body/> element contains the message, usually a request for service or a response to such a request.

WSDL

WSDL, or Web Services Description Language, is used by a server to publish the services that it offers. WSDL is a service description language that gives the service name, location, functions, and specific information about how to bind to and use the service.

UDDI

UDDI, or Universal Description, Discovery and Integration, is a protocol that is used by Web services clients to discover a particular service. UDDI creates a directory of services. UDDI servers can federate to create directories that know about more than just the local services published to the local UDDI server.

Transport

The method that is used for message exchange is called “transport.” The SOAP specification does not specify how messages are transported, but practically speaking, SOA-based implementations that are built with SOAP will send their messages to each other in one of a few ways.

The most common transport mechanism for SOAP messages is HTTP, the protocol of the Web. HTTP is a synchronous, request/response protocol and hence, the easiest message exchange pattern to implement in HTTP is request/response. Other types of message exchange patterns can be built on top of HTTP with some effort.

Another common transport mechanism for SOAP messages is an asynchronous messaging platform such as Sun’s JMS or IBM’s MQSeries. Both of these provide full-featured messaging platforms that will performs request/response,

publish and subscribe, or broadcast messaging with guaranteed, once-only, or in-order options.

Of course, since SOAP doesn't specify how transport happens, an SOA-based system architect is free to specify whatever works best. This means that SOAP messages could be exchanged using SMTP or FTP if that were convenient. Also, new transport mechanisms that are developed in the future can be used when advantageous.

Acknowledgements

Martin Jensen of Oracle suggested changes to the text and contributed the section on "SOA Missteps."

The table comparing loosely and tightly coupled decisions is from http://www.freeroller.net/page/ceperez/20030628#principles_of_loosely_coupled_api (accessed September 2003)

The implementation principals for loose coupling are based on an article by Bill de Hora found at http://www.dehora.net/journal/archives/000300.html (accessed September 2003)